UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Doble Grado en Ingeniería Informática y Matemáticas

TRABAJO FIN DE GRADO

Redes neuronales profundas para reconocimiento de eventos acústicos

Javier Darna Sequeiros
Tutor: Doroteo Torre Toledano

JUNIO 2018

REDES NEURONALES PROFUNDAS PARA RECONOCIMIENTO DE EVENTOS ACÚSTICOS

AUTOR: Javier Darna Sequeiros
TUTOR: Doroteo Torre Toledano

Grupo Audias: Audio, Data Intelligence and Speech
Dpto. Tecnología Electrónica y Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid
junio de 2018

Resumen (castellano)

Este Trabajo Fin de Grado tiene como objetivo la realización de unas primeras pruebas de reconocimiento y clasificación de audio sobre Audio Set mediante redes neuronales. Audio Set es una base de datos de pistas de audio publicada por Google en 2017. Sus datos de entrada están representados como secuencias de un máximo de 10 vectores de longitud 128, cada uno representando un segundo de audio y cuyos elementos son números enteros entre 0 y 255. Las clases están representadas por vectores de longitud variable indicando las clases presentes en cada ejemplo mediante números entre 0 y 526, es decir que hay 527 clases distintas. Cabe destacar sin embargo un marcado desequilibrio en la distribución de las clases, problema en parte compensado por la disponibilidad de dos conjuntos de entrenamiento: uno equilibrado pero con menos datos y otro no equilibrado pero con todos los datos, además de un conjunto de evaluación. Por otro lado comparamos la efectividad de varios perceptrones, una red convolucional y una red LSTM al ser entrenadas con los datos de Audio Set, definiendo antes las métricas “mean Average Precision (mAP)” y “porcentaje de Sonidos Reconocidos (pSR)” o porcentaje de verdaderos positivos. En el caso de los perceptrones, también se ha experimentado con la codificación de los datos, concretamente binaria o bipolar, y el uso de ambos conjuntos de entrenamiento, mientras que en la red convolucional y la red LSTM sólo se usó codificación binaria y el conjunto de entrenamiento equilibrado. Una vez realizadas las pruebas, la red que proporcionó mejores resultados fue la red LSTM con una mAP de 0.24407 y un pSR de 0.30210, demostrando que esta arquitectura de red es la más adecuada para este problema respecto a las arquitecturas que se han probado. Sin embargo estos resultados han resultado inferiores a los de la baseline establecida por Google, con una mAP de 0.314.

Abstract (English)

This Bachelor Thesis’ objective is to perform several audio recognition and audio classification tests over Audio Set by the use of neural networks. Audio Set is a database of sound clips published by Google in 2017. Its data is represented as a sequence of up to 10 vectors of length 128, each one representing one second of audio with integers from 0 up to 255. The target data is represented as vectors with variable length representing which classes are present in each example with numbers from 0 up to 526, implying there are 527 different classes. It must be considered an important unbalance regarding the distribution of the classes along the examples, problem partially solved with the availability of two training sets: one balanced set but with fewer data and one unbalanced set with all the data, all of this in addition to an evaluation set. Apart from this we compared the performance of several multi layer perceptrons, a convolutional network and a LSTM network all of them trained with Audio Set data, defining beforehand the “mean Average Precision (mAP)” and the “Recognised Sounds proportion (pSR)”, which is the proportion of true positives. When training the perceptrons, other experiments were made by changing the coding of the data to binary or bipolar, and by using both training sets, whereas the convolutional neural network and the LSTM network only used binary data with the balanced training set. After the test were made, the network with the best results was the LSTM network with a mAP of 0.24407 and a pSR of 0.30210, proving that this architecture is the most fit for this problem from all of the networks tested. However the results happened to be still inferior to those established by Google’s baseline, which has a mAP of 0.314.

Palabras clave (castellano)

Audio Set, red neuronal, reconocimiento de eventos acústicos.

Keywords (inglés)

Audio Set, neural network, audio event recognition.

Agradecimientos

Quiero dar las gracias:

- A mis padres Rafael y Macarena, y a mi hermana Beatriz por creer en mí siempre que yo no he sido capaz de hacerlo.
- A todos mis amigos por hacer que nunca me sienta solo.
- Al grupo Audias por ayudarme a profundizar en el tema de las redes neuronales y por proporcionarme el material necesario para el desarrollo de este TFG, así como a mi tutor por guiarme durante este.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 Motivación

Esta memoria de TFG pretende documentar el trabajo realizado durante el desarrollo del TFG “Redes neuronales profundas para el reconocimiento de eventos acústicos” durante el segundo cuatrimestre del curso 2017-2018.
Este TFG viene motivado por el problema de reconocimiento de eventos acústicos. Este es un problema de aprendizaje automático en el que se deben entrenar modelos predictivos tales que, dada una pista de audio o una interpretación de esta, puedan reconocer determinados sonidos previamente definidos como clases. Ejemplos de estas clases pueden ser una voz, el motor de un vehículo o un determinado instrumento musical. La creación de modelos capaces de reconocer sonidos tiene bastantes aplicaciones, entre las cuales podemos encontrar ser un sistema para personas con capacidad auditiva limitada que liste mediante texto los sonidos que ocurran alrededor del usuario, o su uso como fase previa para un sistema de procesamiento de audio de carácter general.
El problema de reconocimiento de eventos acústicos ha sido estudiado durante varios años, sin embargo en 2017 Google publicó una base de datos conocida como Audio Set [1]. Con más de 2 millones de ejemplos y 527 clases, este base de datos no tiene precedente en este problema. Este TFG pretende por lo tanto estudiarla.

1.2 Objetivos

En este TFG se pretenden cumplir 3 objetivos principales:

- Estudio de la base de datos en sí. Haremos observaciones, entre otros, sobre los datos de entrada y de salida, la distribución de los ejemplos y los formatos en los que los datos están representados.
- Definición de métricas específicas. Dada la naturaleza del problema, no podemos usar el porcentaje de acierto o de fallo para evaluar los modelos, por lo que tendremos que definir métricas que podamos utilizar en este tipo de problema.
- Comparación de diversas arquitecturas de redes neuronales entrenadas con Audio Set. Se definirán varias arquitecturas de redes neuronales, se entrenarán con los datos de Audio Set y se usarán las métricas definidas para comparar los resultados.

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- **Estado del arte**, donde se dará información de contexto sobre redes neuronales, el problema de reconocimiento de eventos acústicos y Audio Set.
- **Diseño**, donde se estudiará la base de datos, se definirán las métricas y se diseñarán las arquitecturas que se entrenarán con Audio Set.
- **Desarrollo**, donde se comentarán las herramientas elegidas para implementar los modelos y se darán detalles sobre los programas que se utilizarán para realizar las pruebas.
- **Integración, pruebas y resultados**, donde se explicarán las pruebas, se mostrarán los resultados y se discutirán estos últimos.
- **Conclusiones y trabajo futuros**, donde concluiremos sobre los tres objetivos establecidos y discutiremos maneras de progresar en el problema más allá de lo que abarca este TFG.

2 Estado del arte

2.1 Redes neuronales

En el desarrollo de este TFG se han usado distintos tipos de redes neuronales para resolver el problema de reconocimiento de eventos acústicos planteado. Por ello es conveniente introducir conceptos de neurocomputación, así como las arquitecturas que se utilizarán.

2.1.1 Conceptos básicos

La neurocomputación es una rama del aprendizaje automático inspirada por el funcionamiento del cerebro. Consiste en la agrupación de varias unidades capaces de hacer cálculos sencillos de forma que en conjunto puedan hacer cálculos complicados, de forma similar al comportamiento conceptual de las neuronas (de hecho a las unidades se las suele llamar “neuronas”). Estas neuronas realizan un cálculo sencillo sobre los valores que reciben de otras neuronas (por ejemplo una suma ponderada), aplican al resultado una función conocida como “función de activación”, y transmiten el resultado a otras neuronas con las que estén conectadas. Además el sistema puede aprender aumentando o reduciendo los pesos de las conexiones entre las neuronas mediante diversos algoritmos. En comparación a otros tipos de sistemas, las redes neuronales se caracterizan por requerir un tiempo de aprendizaje elevado a cambio de tener un tiempo de predicción reducido, además de por ser muy flexibles en cuanto a su implementación, como veremos más adelante.

Las redes neuronales se pueden usar para un gran número de problemas, sin embargo en el que han tenido más éxito es en el de clasificación. En este tipo de problema se presentan datos a la red para los clasifique en una o varias categorías. El aprendizaje consistirá en mostrarle ejemplos ya clasificados para que intente predecirlos, corrigiendo el peso de las conexiones cuando se equivoque.

Las redes que usaremos estarán compuestas de conjuntos de neuronas llamadas “capas” conectadas secuencialmente. La primera capa, que recibe los datos directamente, se la llama “capa de entrada”. No suele tenerse en cuenta al contar el número de capas de una red ya que no realiza ningún cálculo. La última capa, que muestra el resultado final, se la conoce como “capa de salida”. El resto de capas se llaman “capas ocultas”. Ahora veremos los tipos de redes que usaremos. Generalmente todas las neuronas de una misma capa son del mismo tipo y tienen la misma función de activación.

2.1.2 Funciones de activación

Las funciones de activación usadas por las neuronas juegan un papel fundamental en el rendimiento de una red neuronal. Estas afectan notablemente a factores como los valores que pueden tomar las soluciones, el tiempo de entrenamiento, o incluso los problemas que pueden ser resueltos por la red. Es especialmente importante que las funciones de activación no sean lineales (es decir que no se cumpla $f(ax + by) = af(x) + bf(y)$ para todos x, y, a y b), ya que esto hace que la función calculada por toda la red sea lineal (la composición de funciones lineales es lineal) y que por lo tanto no pueda resolver problemas de clasificación no separables linealmente, categoría en la que entran la gran mayoría de problemas de clasificación incluyendo el que planeamos estudiar en este TFG. A continuación listamos las funciones relevantes para este TFG:

- Sigmoide binaria:

$$f(x)=\frac{1}{1+e^{-x}}$$

Esta función tiene un comportamiento parecido al lineal cerca de x=0, tiene una asíntota horizontal en 1 para valores positivos y otra en 0 para valores negativos, lo que hace que la función tome valores entre 0 y 1.

- Tangente hiperbólica:

$$f(x)=\tanh(x)=\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

Esta función es muy similar a la sigmoide binaria, salvo que toma valores entre -1 y 1 en vez de entre 0 y 1.

- ReLU:

$$f(x)=max(0,x)$$

Un problema que comparten la sigmoide binaria y la tangente hiperbólica es su alto coste computacional. La función ReLU resuelve este problema al tener un coste muy reducido, acelerando el proceso de entrenamiento. Se suele usar como fucnión de activación en las capas ocultas.

2.1.3 Perceptrón

El perceptrón es un tipo de red compuesto de neuronas del mismo tipo también llamadas perceptrones. Estas neuronas son el tipo más simple entre los que veremos. Simplemente realiza la suma ponderada de sus entradas, aplica la función de activación y transmite el resultado obtenido según la siguiente fórmula:

$$Y=f(x\cdot w)$$

Siendo x el vector de entrada, w el vector de pesos, f la función de activación y Y el vector de salida. "·" representa el producto escalar de dos vectores.

No por ser simple es menos útil, ya que una red con dos capas de perceptrones (este tipo de red se conoce como perceptrón multicapa o MLP) teóricamente puede aproximar cualquier función. Las capas de salida de cualquier red están casi siempre compuestas de perceptrones, y además se suelen usar como complemento de otros tipos de capas para procesar en más detalle la información devuelta por estas.

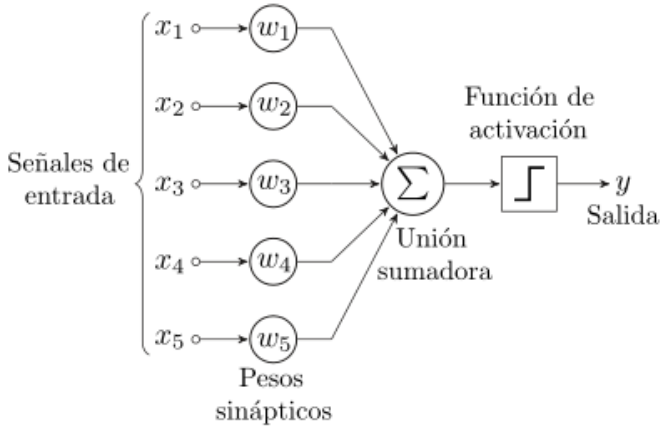

Figura 1: Diagrama de una unidad de perceptrón [12]

2.1.4 Redes convolucionales

Las redes convolucionales se componen de varios tipos de capas distintos: de convolución o extracción de características y de "pooling" o reducción de muestreo.

Las capas de extracción de características se usan frecuentemente como entrada para datos de más de una dimensión (por ejemplo imágenes), así como para reducir el tamaño de los datos de forma a hacerlos más fáciles de tratar. Los datos van siendo recorridos por una ventana llamada “kernel” que extrae una pequeña parte de estos bajo forma de matriz. A esta matriz se le aplican un número de filtros predeterminado, siendo cada uno una operación de convolución por unos valores correspondientes al filtro y que se van entrenando, y se envía el resultado como salida. Debido al tamaño del kernel, la salida suele ser más pequeña que la entrada, aunque no es necesario que sea así.
Detrás de una capa de convolución suele ir otra capa de convolución o una capa de reducción de muestreo. Estas últimas de nuevo extraen submatrices de la capa anterior, y por cada una devuelve un único valor según un criterio que suele ser la media de los valores o el máximo.
Detrás de las capas de pooling pueden ir más bloques de capas de convolución y pooling o capas de perceptrón.

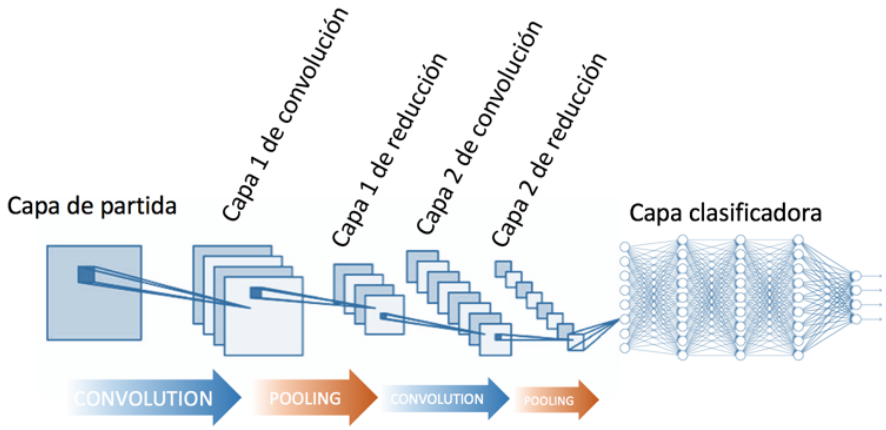

Figura 2: Funcionamiento de una red CNN [13]

2.1.5 Redes recurrentes y LSTM

En las redes vistas hasta ahora la información viaja de la capa de entrada a la de salida en una dirección. Las redes recurrentes se caracterizan por tener bucles a través del tiempo al conectar algunas neuronas consigo mismas, lo que las hace especialmente útiles para analizar secuencias temporales de datos. Sin embargo estas redes sufren un problema de desvanecimiento de gradiente. La acumulación de información a lo largo del tiempo acaba acercando a las nuevas entradas a regiones de la función de activación cuyo gradiente es cercano a 0, haciendo que la sensibilidad de la red a los valores de entrada decaiga, limitando su capacidad de aprendizaje. Este problema fue solucionado con la introducción de las redes LSTM (long short term memory) cuyas neuronas contienen cada una una célula de memoria que les permite aprender incluso secuencias largas sin que las nuevas entradas pierdan relevancia.
Al calcular una salida, la neurona utiliza esta salida para entrenar una célula de memoria que participará en el cálculo de posteriores salidas, así como para entrenar tres puertas que controlan la influencia de la entrada y de la memoria en el cálculo de la salida. Estas puertas son:

- La puerta de entrada, que determina la influencia del valor de entrada en el cálculo de la salida.
- La puerta de salida, que determina la influencia de la memoria en el cálculo de la salida.

- La puerta de olvido, que determina en qué grado la memoria se ve influenciada por la salida más reciente.

A cada vector de la secuencia, la neurona realiza las siguientes actualizaciones:

$$f_{t+1} = \sigma(W_f x_{t+1} + U_f h_t + b_f)$$
$$i_{t+1} = \sigma(W_f x_{t+1} + U_f h_t + b_f)$$
$$o_{t+1} = \sigma(W_o x_{t+1} + U_o h_t + b_o)$$
$$c_{t+1} = f_{t+1} \circ c_t + i_{t+1} \circ \tanh(W_c x_{t+1} + U_c h_t + b_c)$$
$$h_{t+1} = o_t \circ \tanh(c_t)$$

Siendo las variables:

- x: vector de entrada.
- f: vector de activación de la puerta de olvido.
- i: vector de activación de la puerta de entrada.
- o: vector de activación de la puerta de salida.
- h: vector de salida de la unidad LSTM.
- c: vector de estado de la célula de memoria.
- W, U: matrices de pesos entrenables.
- b: valor de sesgo.
- σ : función sigmoide binaria
- $\circ$: producto componente a componente

Las capas LSTM pueden devolver una secuencia de vectores (por ejemplo para introducirla en otra capa LSTM), o sólo el último vector (por ejemplo para representar la salida).

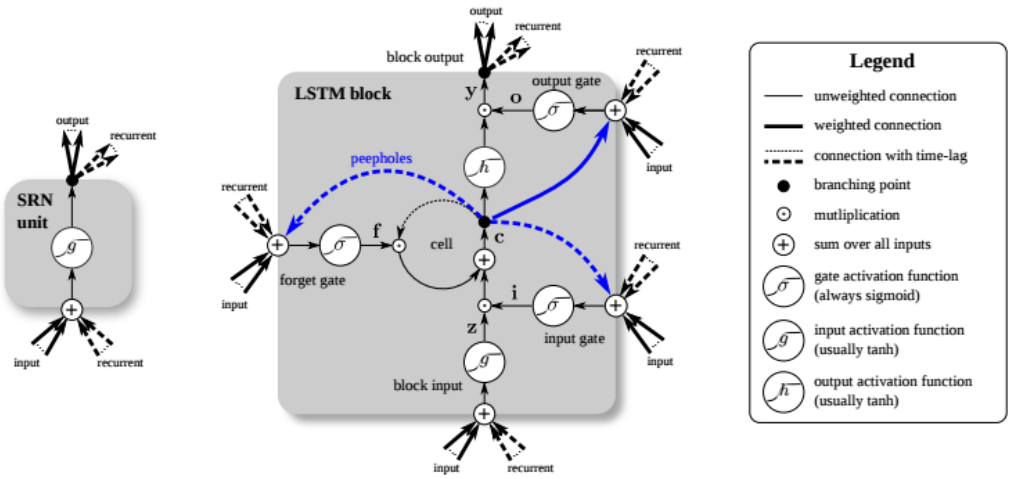

Figura 3: Estructura de una una neurona recurrente simple (izquierda) y de una neurona LSTM (derecha) [14]

2.2 Antecedentes

El problema de reconocimiento de eventos acústicos ya estaba siendo estudiado antes de la publicación de Audio Set. A continuación comentaremos brevemente varios estudios que han sido realizados con objetivos iguales o similares.

2.2.1 CLEAR

CLEAR [2] fue un estudio realizado en 2006 por varias universidades interesado en sonidos producidos por actividades humanas, tales como voces, aplausos, o sonidos de objetos cotidianos como puertas. Junto con el reconocimiento de eventos acústicos, trabajaron también con la detección de eventos acústicos que exige, además de reconocer determinados sonidos, localizarlos en el tiempo. En la evaluación el equipo creó varias bases de datos con varios cientos de ejemplos, concretamente dos cuyos ejemplos eran sonidos aislados, y una formada a partir de grabaciones de seminarios. Los ejemplos podían pertenecer a 14 clases distintas, aunque sólo se evaluaron 12. Para modelar los datos se usaron dos sistemas basados en modelos ocultos de Markov (HMMs) y otro basado en máquinas de soporte vectorial (SVMs), modelos que no se tratarán en este TFG.

2.2.2 Urban sound taxonomy

Este estudio [3] fue publicado en 2014 por un equipo de la universidad de Nueva York. En este se creó una base de datos de 27 horas de audio en ejemplos para clasificar entre 10 clases de sonidos urbanos como la bocina de un coche o el ladrido de un perro, seleccionadas como subconjunto de una taxonomía más grande. Sin embargo, para facilitar el entrenamiento de modelos, también se creó una base de datos reducida llamada UrbanSound8K, con 8732 ejemplos de duración inferior o igual a 4 segundos y 10 clases. Para establecer una baseline, se hicieron pruebas con 5 algoritmos de aprendizaje automático: árbol de decisión, 5 vecinos próximos, random forest, máquinas de soporte vectorial, y majority vote classifier.

2.2.3 DCASE 2016 challenge

El DCASE 2016 [4] fue una competición de aprendizaje automático orientada al reconocimiento de eventos acústicos. Esta se dividió en 4 categorías: identificar un lugar a partir de audio grabado ahí, detección de eventos acústicos, reconocimiento de eventos acústicos a partir de audio grabado en entornos reales, y etiquetado de sonidos en entornos domésticos. Se creó una base de datos para cada categoría, y los participantes debieron entrenar modelos con esas bases de datos y publicar papers con sus resultados. La base con más clases tiene 18 clases y la más poblada tiene 10 horas de audio.

2.3 Audio Set

Como se puede observar, los antecedentes descritos previamente son bastante específicos en cuantos a los sonidos a identificar. Además el tamaño de la base de datos y el número de clases suelen ser ambos bastante reducidos. Estos fueron los motivos que impulsaron a Google a publicar Audio Set.

2.3.1 Descripción de Audio Set

Audio Set es una base de datos publicada por Google en 2017 para impulsar el problema de reconocimiento de eventos acústicos. Además de tener una gran variedad de sonidos para reconocer, con más de 500 clases, el número de ejemplos es mucho mayor que en los otros casos, con más dos millones de ejemplos. Esta gran cantidad de ejemplos ha sido recopilada a partir de vídeos de Youtube, de los cuales se han extraído 10 segundos de audio, lo que hace que en total Audio Set cuente con más de 5000 horas de audio. Al provenir de vídeos que no han sido grabados para este propósito, los sonidos no están aislados y pueden solaparse.

Las 527 clases están organizadas bajo una estructura jerárquica con profundidad máxima 6, siendo los nodos más altos categorías de sonidos y los más bajos sonidos específicos. Cada ejemplo puede estar etiquetado como una o más clases.
Los datos están disponibles en dos formatos. El primero contiene los enlaces a los videos de forma que se puede acceder directamente al audio para tratarlo con libertad, mientras que el segundo contiene las salidas de una red de embedding con la que Google ha tratado previamente el audio, de modo que se puedan entrenar modelos sin necesidad de procesar señales de audio.

2.3.2 Baseline de Google

Junto con la publicación de la base de datos, Google entrenó un modelo con el fin de establecer un punto de referencia para los modelos que se creen posteriormente [5]. La información sobre el modelo en sí es bastante escasa, explicando unicamente que usaron un perceptrón multicapa con una capa oculta sobre 485 de las 527 clases originales y sus resultados bajo la forma de varias métricas, de las cuales nos interesaremos en una en concreto: la mAP (mean Average Precision), de la que hablaremos en detalle más adelante. La baseline de Google tiene una mAP de 0.314, y los estudios realizados sobre Audio Set suelen compararse con este resultado.

2.3.3 Estado del arte de Audio Set

Debido a que Audio Set sólo tiene un año, los progresos realizados sobre esta taxonomía han sido limitados, sin embargo ciertas arquitecturas han empezado a establecerse como estado del arte. Estas son las redes recurrentes, lo cual no es de extrañar ya que los datos de embedding forman una serie temporal, para los que estas arquitecturas suelen ser bastante efectivas.
Entre estas destaca la red que posee el mayor acierto actualmente, con un mAP de 0.327. Esta red sigue una arquitectura de la que no hemos hablado conocida como clasificador con modelo de atención probabilista [6]. Es un tipo de red recurrente que aplica el mismo clasificador a cada segmento de tiempo de un ejemplo, para luego integrar los resultados individuales en un resultado final. Sin embargo, al mismo tiempo que estos clasificadores se va entrenando una función que determina el peso de cada segmento dentro del cálculo final en función de la entrada. Se lo llama probabilista porque los pesos que devuelve la función están calculados de forma que sumen 1, de manera que la función sea una medida de un espacio de probabilidad.

3 Diseño

3.1 Estudio de los datos

El primer objetivo en el que nos centraremos será el estudio de los datos con los que vamos a trabajar. En este TFG sólo nos interesarán los datos en su formato de embedding, ya que los resultados con los que vamos a comparar nuestros resultados usan este formato y el procesamiento de señales de audio está fuera de lo que pretendemos abarcar con este trabajo.

Al intentar descargar la base de datos, podemos ver que esta esta dividida en tres subconjuntos: entrenamiento equilibrado, entrenamiento no equilibrado y evaluación. El motivo por el que hay dos conjuntos de entrenamiento distintos es, según se puede ver en la página de Audio Set, que la distribución de las clases a lo largo de los ejemplos está seriamente desequilibrada: por ejemplo, la clase “Music” está presente en el 56% de los ejemplos, mientras que otras clases apenas aparecen representadas por ejemplos. Por otro lado, el conjunto de entrenamiento equilibrado posee un número de ejemplos muy inferior al del conjunto no equilibrado, concretamente unos 22000 ejemplos contra 2 millones. Cada uno de los dos conjuntos de entrenamiento tiene ventajas e inconvenientes respecto al otro, por ello, se entrenarán modelos para ambos de forma que se puedan comparar los resultados que ofrezcan.

Los datos de embedding están disponibles en la página de Audio Set en formato .tfrecord. Este es un formato exclusivo de Tensorflow, lo cual es de esperar ya que Tensorflow pertenece a Google. Cada conjunto de datos está en un directorio con una cantidad de archivos que puede superar los 4000. Los vectores de clases son devueltos como un tensor de longitud variable con las clases presentes en cada ejemplo representadas por números entre 0 y 526. Los datos por otro lado son devueltos como una secuencia de vectores de longitud 128, siendo cada elemento de cada vector un numero entero entre 0 y 255. Generalmente los ejemplos representan segmentos de audio de 10 segundos, representando cada vector 1 segundo, sin embargo esto no se cumple si la duración del vídeo del que proceden es inferior a 10 segundos, lo que hace que estas secuencias de vectores tengan longitud variable inferior o igual a 10 segundos. Esto no es relevante para una red recurrente, ya que pueden recibir secuencias de longitud arbitraria, sin embargo es un problema serio para otras redes como los MLP o las CNN (redes convolucionales), ya que esperan entradas de dimensión fija. La opción elegida para resolver este problema fue añadir vectores con valor 0 en todas sus componentes de forma que todas las entradas tuvieran 1280 valores.

Sin embargo, esta solución fue imposible de implementar en los archivos originales. El formato .tfrecord está pensado para usarse directamente en modelos de Tensorflow sin necesidad de leer los datos. Esto es muy útil cuando se planea usar una base de datos propia o una que apenas requiera manipular los datos, sin embargo si se pretenden tratar de forma significativa los datos (por ejemplo expandir algunos datos para que tengan una dimensión consistente), este formato trae severas complicaciones. Para empezar, como hemos comentado previamente, .tfrecord es un formato exclusivo de Tensorflow, lo cual obliga a elegir entre convertir los datos en tiempo de ejecución a un formato legible por otros modelos (por ejemplo Numpy) o usar Tensorflow para definir los modelos. Se empezó optando por la primera opción, ya que se planeaba usar la librería Keras para el desarrollo de los modelos. Desarrollar el código necesario para convertir los datos requirió una gran cantidad de tiempo de búsqueda. Esto se debe a que la información sobre como extraer datos legibles de un archivo .tfrecord es muy escasa al no ser este el uso

intencionado de este formato y que la estructura interna de los datos no venía explicada de forma suficientemente rigurosa. Una vez obtenido el código, este resultó ser lento hasta extremos inaceptables, debido a la gran cantidad de conversiones individuales requeridas, lo que finalmente obligó a descartar la idea y optar por el uso de modelos de Tensorflow. La propia conversión no resultó ser un problema, ya que Tensorflow cuenta con funciones para convertir directamente modelos de Keras a Tensorflow, sin embargo para realizar las manipulaciones deseadas se requerían conocimientos de manipulación de tensores, clase básica de datos de Tensorflow cuyo cálculo es muy eficiente por las funciones de la librería pero no es humanamente legible, lo que los hace complicados de manipular. Todo este proceso empezó a comprometer seriamente los plazos de desarrollo del TFG, por lo que el uso de los datos en formato .tfrecord acabó siendo descartado por completo.
Sin embargo, en el grupo de usuarios de Google+ "Audio Set users" [7], un usuario convirtió los datos a formato .h5, que fueron finalmente los que se usaron para las pruebas. El archivo descargado incluye un archivo en formato .h5 para cada conjunto de datos. Las clases están representadas como un vector booleano de longitud 527, representando para cada clase True su presencia y False su ausencia. Los datos están representados como una matriz de 10 filas por 128 columnas de números enteros entre 0 y 255. En los casos problemáticos previamente mencionados, las matrices habían sido rellenadas con ceros, lo que convenientemente resulta ser lo que se planeaba hacer con los datos.
Además de los archivos con los datos, viene incluido un archivo "readme" con varias funciones útiles. La primera permitía extraer datos del archivo bajo la forma de un array de numpy. Esto ofrece otra ventaja sobre el formato .tfrecord, ya que ofrece más libertad a la hora de escoger librerías con las que crear los modelos. La segunda función transforma los datos a números con coma flotante, y además los normaliza con la fórmula (v – 128) / 128, de forma que tomen valores entre -1 y 1. La normalización ya había sido considerada, pues las redes tienden a ser más eficaces con valores normalizados, por lo que se decidió usar este función para ello. La tercera función simplemente convierte los datos booleanos a los enteros 0 y 1, lo cual es necesario ya que las redes neuronales funcionan con números.

3.2 Métricas utilizadas

Una característica que distingue a Audio Set de la mayoría de problemas que involucran redes neuronales es el hecho de que es multietiqueta, es decir que un único ejemplo puede contener más de una clase al mismo tiempo, lo que supone un problema a la hora de evaluar su efectividad. En primer lugar no podemos usar la entropía categórica cruzada, función de pérdida ampliamente utilizada en problemas de clasificación, ya que esta exige en los vectores como máximo una clase activa a la vez. No se puede usar el clásico porcentaje de acierto o de fallo porque, en este caso, el grado de acierto de un ejemplo no es una función binaria. Es por ello que debemos definir métricas que se adapten a este tipo de problema y que permitan evaluar los modelos que creemos.

3.2.1 Media de aciertos

Este fue el primer intento de crear una métrica que funcionara en problemas multietiqueta. Consiste en una simple generalización del concepto de porcentaje de acierto, dividiendo el número de clases acertadas por el número total de clases. Esta métrica demostró ser bastante ineficaz debido al elevado número de clases de las cuales la gran mayoría tendrán valor 0. Esto da lugar a casos como que una red que siempre prediga 0 para todas las clases tenga una media de aciertos superior a 0.9, la cual es una evaluación bastante cuestionable de la efectividad de la red. Por estos motivos esta métrica no se usará en los experimentos.

3.2.2 mAP (mean Average Precision)

La AP (average precision) es una métrica muy utilizada en modelos cuyos resultados no se pueden definir simplemente como “acierto” o “fallo”. Sin embargo esta métrica tiene el problema de que carece de definición objetiva, problema que además se acentúa por el hecho de que Google no ha descrito en la página web ni en el paper la fórmula utilizada. Aún así, a partir de varias de las definiciones más utilizadas y el tipo de problema, podemos establecer una definición de la mAP que usaremos para evaluar los modelos. El principal problema de la media de aciertos es que, dada la gran cantidad de componentes con valor 0 en cada ejemplo, la métrica dará valores muy altos independientemente de las clases predichas por el modelo siempre que sean pocas. Podemos resolver este problema simplemente ignorando las componentes en las que “la red no ha hecho nada”, es decir los casos en los que tanto la componente predicha como la del vector objetivo valen 0. Concretamente esto nos daría la fórmula siguiente:

$$AP = \frac{\sum_i p_i t_i}{\sum_j p_j + t_j}$$

Siendo p el vector predicho, t el vector objetivo y + la suma booleana componente a componente.
Cada par de vectores devolvería un valor, siendo la mAP la media de todos ellos. Esta será nuestra métrica principal y la que usaremos para evaluar los modelos.

3.2.3 pSR (porcentaje de sonidos reconocidos)

Esta métrica adicional, también conocida como “Recall”, se obtiene calculando el porcentaje de aciertos de la predicción, pero únicamente sobre las clases presentes del ejemplo, ignorando las predicciones realizadas en las clases ausentes. Esta métrica, aún siendo menos reveladora que la mAP, es útil para medir la capacidad de los modelos de reconocer los verdaderos positivos (true positives). Su fórmula sería la siguiente:

$$pSR = \frac{\sum_{i,j} p_{ij} t_{ij}}{\sum_{k,l} t_{kl}}$$

Siendo (p_{ij}) la matriz formada por los vectores predichos y (t_{ij}) la matriz formada por los vectores objetivo.

3.3 Diseño de las redes

Una vez estudiados los datos y definidas las métricas, podemos diseñar los modelos que usaremos para realizar las pruebas. Usaremos 3 arquitecturas distintas: un MLP, una CNN y una red LSTM. Además, en el caso del MLP experimentaremos con el número de capas, el formato de entrada, y los dos conjuntos de entrenamiento. Las otras arquitecturas sólo se probarán con los datos sin modificar y únicamente con el conjunto de entrenamiento equilibrado.

3.3.1 Características comunes

Aún usando varias arquitecturas distintas, hay ciertos parámetros que se usarán en todos los modelos:

- El optimizador utilizado será un Adam, ya que funciona en casi todos los casos y no requiere establecer un alfa.

- Generalmente se usa la entropía categórica cruzada como función de pérdida en los modelos de clasificación. Sin embargo nuestro problema no lo permite por ser multietiqueta, por lo que, salvo una excepción, todos los modelos usarán como función de pérdida la entropía binaria cruzada.
- Todas las unidades salvo las de la capa de salida usarán ReLU como función de activación.
- La capa de salida será una capa de perceptrón (también llamada “densa” o “fully connected”) con 527 unidades. Obviamente cada una representa una clase.
- Generalmente las neuronas de la capa de salida usan “softmax” como función de activación. Sin embargo, el hecho de que las salidas deban sumar 1 causa problemas cuando hay más de una clase activa al mismo tiempo. Por ello, salvo una excepción, la capa de salida de todos los modelos usará como función de activación la sigmoide binaria.

Una vez aclarados estos puntos, podemos definir cada modelo específicamente.

3.3.2 Perceptrón multicapa

Al ser la arquitectura de red neuronal más versátil, usaremos los siguientes modelos basados en MLP:

- MLP con una capa oculta con el conjunto de entrenamiento equilibrado.
- MLP con dos capas ocultas con el conjunto de entrenamiento equilibrado.
- MLP con tres capas ocultas con el conjunto de entrenamiento equilibrado.
- MLP con una capa oculta con el conjunto de entrenamiento equilibrado, pero con datos de salida bipolares en vez de binarios. Este caso se ha considerado ya que algunas arquitecturas de redes neuronales se benefician de este tipo de codificación.
- Todos los anteriores con el conjunto de entrenamiento no equilibrado.

Los cuatro modelos compartirán estas características:

- La capa de entrada tendrá 1280 unidades, recibiendo como entrada la matriz de 10x128 aplanada aplanada a un vector de longitud 1280.
- Todas las capas ocultas tendrán 1500 unidades.
- Detrás de cada capa oculta habrá una capa “dropout” con probabilidad 0.3.

El último modelo contará con las siguientes características exclusivas:

- La función de pérdida será el error cuadrático medio.
- Las neuronas de la capa de salida usarán la tangente hiperbólica como función de activación.

3.3.3 Red convolucional

Al tomar los datos la forma de una matriz, una red convolucional debería poder extraer características de esta al igual que hace con imágenes. Esta red se compondrá de la siguiente forma:

- La primera capa será una capa de convolución con 16 filtros y un kernel de dimensión 3x1. Recibirá las entradas como una matriz de tamaño 10x128.
- La segunda capa será una capa de reducción de muestreo mediante máximo con ventana de dimensión 2x2. Tras esta capa se añadirá una capa “dropout” con probabilidad 0.3.
- La última capa oculta será una capa densa de 600 unidades.

No se han añadido más capas de extracción de características para evitar reducir demasiado las dimensión de los datos. Por otro lado se ha elegido un kernel unidimensional porque los datos, al provenir de un embedding, carecen del significado local en el espacio que podrían

tener por ejemplo los píxeles de una imagen, mientras que sabemos que tienen significado local en el tiempo, al representar cada fila un segundo.

3.3.4 Red LSTM

Al ser los datos una secuencia temporal de vectores, una red LSTM debería ser bastante eficaz, por lo que crearemos un modelo con la siguiente estructura:

- La primera capa será una capa LSTM con 600 unidades y que devolverá un único vector. Las entradas serán recibidas como secuencias de 10 vectores de longitud 128. Tras esta capa se añadirá una capa “dropout” con probabilidad 0.3.
- La última capa oculta será una capa densa con 600 unidades.

Originalmente se iban a usar dos capas LSTM, la primera devolviendo una secuencia de vectores para usar en la segunda. Sin embargo la red resultó no ser capaz de aprender, por lo que se rectificó al modelo actual.

4 Desarrollo

4.1 Herramientas utilizadas

Una vez realizadas todas las consideraciones de diseño necesarias, debemos implementar los modelos para realizar las pruebas. Los programas se realizarán en lenguaje Python, en su versión 2.7.12, con las siguientes herramientas.

4.1.1 Keras

Keras [8] es una librería para la implementación de redes neuronales a alto nivel, orientada al prototipado rápido y a la experimentación. Esta librería posee una API lo suficientemente completa para realizar implementar los modelos diseñados y probarlos en condiciones adecuadas, además de ser bastante fácil de utilizar, lo que la hace una buena opción para el desarrollo de los modelos en sí. Por motivos de compatibilidad, se usará la versión 2.1.3 de Keras, la cual se ha instalado mediante virtualenv.

4.1.2 Tensorflow-GPU

Tensorflow [9] es una librería de aprendizaje automático desarrollada por Google. Esta librería es necesaria para el funcionamiento de Keras, pues requiere un backend que puede ser Tensorflow, Theano o CNTK. De entre las 3 usaremos Tensorflow porque es el backend recomendado en la propia página de Keras.
Por otro lado, las pruebas con redes más complejas y especialmente las que usen el conjunto de entrenamiento no equilibrado corren el riesgo de tardar demasiado tiempo en entrenar, es por ello que, en vez de usar la versión de Tensorlfow para CPU, optaremos por su versión para GPU. El uso de GPUs para entrenar redes neuronales es una práctica muy común, ya que reduce drásticamente el tiempo requerido para el proceso de entrenamiento. Por motivos de compatibilidad se usará la versión 1.4.0 de Tensorflow-GPU, la cual se ha instalado mediante virtualenv.

4.1.3 Nvidia Geforce GTX 1080

Como hemos mencionado previamente, el uso de GPUs aumenta notablemente la velocidad de cálculo a la hora de entrenar redes neuronales. En los experimentos se usará una GPU de modelo Nvidia Geforce GTX 1080.

4.1.4 Cuda

Además de la propias GPUs, se requieren ciertas librerías para que esta pueda ser usada por Tensorflow. Cuda [10] es una arquitectura de Nvidia para cálculo paralelo. Esta arquitectura proporciona un gran incremento en el rendimiento del sistema, que es precisamente lo que necesitamos. Usaremos las versión 8.0.61 de Cuda

4.1.5 CuDNN

CuDNN [11] es la otra librería requerida por Tensorflow para funcionar en GPUs. CuDNN es una librería de Nvidia para redes neuronales profundas, que acelera varias funciones primitivas como convoluciones, pooling o funciones de activación. Usaremos la versión 6.0.0 de CuDNN.

4.2 Programas

Para realizar las pruebas se han desarrollado los siguientes programas:

- MLPAudioSet.py: Este programa entrena un perceptrón multicapa con conjuntos de entrenamiento pequeños y guarda el modelo resultante para su explotación. Los argumentos de programa permiten definir, entre otros, el número de capas, el número de unidades por capa y el tipo de salida (binaria o bipolar).
- MLPAudioSetF.py: Este programa entrena un perceptrón multicapa con conjuntos de entrenamiento de tamaño arbitrario y guarda el modelo resultante para su explotación. Los argumentos de programa permiten definir, entre otros, el número de capas, el número de unidades por capa y el tipo de salida (binaria o bipolar).
- CNNAudioSet.py: Este programa entrena una red convolucional con conjuntos de entrenamiento pequeños y guarda el modelo resultante para su explotación.
- LSTMAudioSet.py: Este programa entrena una red LSTM con conjuntos de entrenamiento pequeños y guarda el modelo resultante para su explotación.
- cargarModelo.py: Este programa carga un modelo guardado previamente, lo evalúa con un conjunto de evaluación y muestra la mAP y el pSR resultantes.

Los 4 primeros programas siguen la misma estructura básica:
- Creación del modelo.
- Carga de los datos de entrenamiento.
- Carga de los datos de validación.
- Entrenamiento.
- Guardado del modelo.

cargarModelo.py sigue la siguiente:
- Carga del modelo guardado.
- Carga de los datos de evaluación.
- Predicción de los datos por el modelo.
- Cálculo de las métricas.

Ahora nos centraremos en cada una de las etapas que no sean triviales.

4.2.1 Creación del modelo

Todos los modelos son modelos “Sequential”, una de las dos formas de crear modelos en Keras. Las capas y sus parámetros han sido añadidos según como se ha especificado en la sección Diseño, al igual que sus parámetros de compilación.

4.2.2 Extracción de datos, preprocesamiento

Para cargar los datos de entrenamiento y evaluación, se ha usado la función descrita en el archivo “readme” incluido junto a la base de datos. Esta función originalmente usa las funciones de la librería h5py para abrir el archivo, extraer los identificadores de los vídeos, el embedding y las etiquetas como referencias al archivo (es decir que los datos no se cargan realmente en este paso) y convertirlos a un array de Numpy para que se puedan usar. En todos los programas se ha modificado la función para que no extraiga los identificadores de los vídeos, ya que son irrelevantes para los presentes experimentos. Una vez extraídos se han aplicado las otras dos funciones descritas en el “readme” para convertir los datos de embedding a reales entre -1 y 1 y para convertir las etiquetas a enteros con valor 0 o 1.
En MLPAudioSet.py y en MLPAudioSetF.py se han redimensionado las matrices para que tengan dos dimensiones en vez de tres. Además se especifica que se quieren usar valores bipolares, las etiquetas se convierten a enteros con valores -1 o 1.

En CNNAudioSet.py se ha añadido una dimensión de tamaño 1 a la matriz para adecuarse a los requerimientos de la capa de convolución utilizada.
Por otro lado, cargar toda la base de datos en memoria puede ser problemático al usar el conjunto de entrenamiento no equilibrado, ya que la matriz resultante tendría $2*10^6*128*10=256*10^7$ números con coma flotante, lo que puede hacer fallar el programa por falta de memoria. Afortunadamente Keras permite entrenar sus modelos con generadores, es decir funciones que a cada llamada devuelven una parte de los datos. Por esto, en MLPAudioSetF.py se ha creado un generador a partir de la función de carga de datos. Este generador, en vez de convertir todos los datos a un array, convierte sólo 128 filas y las devuelve.

4.2.3 Entrenamiento

El entrenamiento será realizado con el método “fit” del modelo salvo en MLPAudioSetF.py, que se ha usado el método “fit_generator”. El entrenamiento durará un máximo de 50 épocas y se hará por lotes de 128 ejemplos, coincidiendo así con el número de ejemplos devueltos por el generador.
Además han sido tomadas medidas para prevenir el sobrentrenamiento:

- El conjunto de evaluación será usado como conjunto de validación, lo que permitirá averiguar fácilmente cuando un modelo empieza a sobrentrenar. Se ha usado el mismo conjunto para la validación y la evaluación para aprovechar el hecho de que el conjunto está equilibrado, sin embargo esto puede sesgar los resultados y por lo tanto es una limitación del TFG.
- Se ha implementado el cálculo de la mAP para que pueda ser usado como métrica durante el entrenamiento, ya que nuestro objetivo es maximizar este valor. Nótese que para la definición de métricas personalizadas se deben manipular los vectores predichos y objetivo usando las funciones de backend de Keras, el cual funciona de forma similar a la manipulación de tensores de Tensorflow. De hecho, al estar usándose Tensorflow como backend de Keras, realmente se están manipulando tensores de Tensorflow, sin embargo al usarse las funciones de backend de Keras, este código debería funcionar también con Theano o CNTK.
- Se ha establecido un callback “EarlyStopping” que monitoriza la mAP con paciencia 3. Esto quiere decir que el entrenamiento terminará si la mAP calculada no mejora durante 3 épocas seguidas, aunque no hayan pasado todas las épocas. Este callback se ha declarado para todos los modelos salvo para la red LSTM debido a que su aprendizaje resultó ser bastante lento e irregular.

4.2.4 Guardado del modelo

Para el entrenamiento se ha creado un callback “ModelCheckpoint” que monitoriza la mAP. Este callback guarda el modelo cada vez que la mAP calculada aumente al final de cada época. Esto asegura que al final del programa queda guardado el modelo en su mejor momento del entrenamiento.

5 Integración, pruebas y resultados

5.1 Desarrollo de las pruebas

Una vez implementados los modelos, podemos realizar las pruebas en las que compararemos la efectividad de varias arquitecturas de redes neuronales clasificando los ejemplos de Audio Set.
En el desarrollo de las pruebas se fijará la semilla del generador de números aleatorios, de forma que la inicialización aleatoria de los pesos de los modelos no interfiera con la consistencia de los resultados.
Las pruebas se realizarán de la siguiente forma:

- Se ejecutarán los programas MLPAudioSet.py, MLPAudioSetF.py, CNNAudioSet.py y LSTMAudioSet.py con el fin de crear y entrenar todos los modelos definidos en las secciones anteriores.
- Se ejecutará el programa cargarModelo.py con cada uno de los modelos creados sobre el conjunto de evaluación. El programa nos devolverá los resultados finales: la mAP y el pSR de cada modelo.

5.2 Resultados

Las pruebas han dado lugar a los siguientes resultados:

Modelo (conjunto entrenamiento, codificación)	**mAP**
MLP con 1 capa oculta (eq, bip)	0.13486
MLP con 1 capa oculta (no eq, bip)	0.19356
MLP con 3 capas ocultas (no eq, bin)	0.20163
MLP con 2 capas ocultas (no eq, bin)	0.20613
MLP con 1 capa oculta (eq, bin)	0.20817
MLP con 1 capa oculta (no eq, bin)	0.20821
CNN (eq, bin)	0.21603
MLP con 2 capas ocultas (eq, bin)	0.22442
MLP con 3 capas ocultas (eq, bin)	0.23180
Red LSTM (eq, bin)	0.24407

Tabla 1: Clasificación de los modelos según su mAP

Modelo (conjunto entrenamiento, codificación)	pSR
MLP con 1 capa oculta (eq, bip)	0.15848
MLP con 1 capa oculta (no eq, bip)	0.22697
MLP con 3 capas ocultas (no eq, bin)	0.23529
MLP con 2 capas ocultas (no eq, bin)	0.24079
MLP con 1 capa oculta (no eq, bin)	0.24166
MLP con 1 capa oculta (eq, bin)	0.24249
CNN (eq, bin)	0.25595
MLP con 2 capas ocultas (eq, bin)	0.27542
MLP con 3 capas ocultas (eq, bin)	0.28706
Red LSTM (eq, bin)	0.30210

Tabla 2: Clasificación de los modelos según su pSR

Leyenda:

- eq: Conjunto de datos equilibrado
- no eq: Conjunto de datos no equilibrado
- bin: Codificación binaria
- bip: Codificación bipolar

5.3 Comparación de los resultados

El primer dato que podemos destacar de las dos tablas es que el orden es idéntico salvo en los casos correspondientes al MLP con una capa oculta y codificación binaria, sin embargo en ambos métricas la diferencia de valores es muy pequeña, por lo que la diferencia de orden no es realmente relevante. Una vez considerado este caso, podemos comparar la efectividad de las distintas arquitecturas independientemente de la métrica utilizada.

Los modelos que han proporcionado peores resultados son, con diferencia, los perceptrones entrenados con salidas en codificación bipolar. Esto puede deberse, más que a los datos en sí, a la adaptación del modelo a estos datos. El error cuadrático medio es una función de pérdida que suele proporcionar peores resultados que la entropía cruzada binaria, y es posible el uso de ReLU como función de activación de las neuronas de la capa oculta haga más difícil a la capa de salida devolver valores negativos. Por otro lado, podemos ver que el uso del conjunto de entrenamiento no equilibrado ha mejorado notablemente los resultados de esta arquitectura.

Esto es sorprendente ya que es la única arquitectura que se ha beneficiado de este conjunto de datos. En los MLP con codificación binaria, el uso del conjunto de entrenamiento no equilibrado ha empeorado los resultados, siendo mayor el efecto cuantas más capas ocultas tenga el modelo (nótese que en el MLP con una capa oculta la diferencia es mínima), sin embargo los resultados siguen siendo mejores que los de los modelos con codificación bipolar. Esto de todas formas puede deberse a la repartición heterogénea de las clases en este conjunto de datos.

En el caso de los MLP usando conjunto de entrenamiento equilibrado, los resultados, como se puede esperar, son mejores cuantas más capas ocultas posee el modelo. Aunque no han resultado ser las arquitecturas óptimas para resolver el problema, se puede apreciar que los perceptrones dan resultados aceptables, demostrando su versatilidad. Por otro lado cabe

destacar que el MLP con 3 capas ocultas es bastante sensible a la semilla aleatoria inicial. Según su valor la red puede no aprender en absoluto. Esto probablemente se deba a que haya algún mínimo local en la función de pérdida que atraiga una cantidad anormalmente grande de puntos.
La red convolucional ha proporcionado mejores resultados que el MLP con una capa oculta, pero peores que el de dos. Dado que esta red tiene dos capas ocultas (si consideramos la convolucional y la de pooling como una sola capa), podemos concluir que esta arquitectura es poco apropiada para este problema. Esto probablemente se deba a que los datos solo tienen proximidad en el tiempo y no en el espacio, lo que limita la efectividad de una red convolucional.
Finalmente la red que ha devuelto los valores más prometedores es la red LSTM, dando mejores resultados que el perceptrón con tres capas ocultas a pesar de tener sólo dos. No es de extrañar ya que estas redes son especialmente efectivas en secuencias temporales, pero esto demuestra que habría que centrar la atención en este tipo de redes si se aspira a resultados mejores.
Por último podemos observar que ninguna de las redes han ofrecido mejores resultados que la baseline establecida por Google. Los motivos para esto pueden ser, en primer lugar, que el modelo se realizó sobre 485 clases, mientras que nuestros modelos se usaron 527, y en segundo que podemos haber realizado una suposición errónea respecto a la arquitectura del modelo creado por Google o respecto a la fórmula de la mAP utilizada.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

A lo largo de esta memoria de TFG, hemos estudiado la evolución del problema de reconocimiento de eventos acústicos con redes neuronales para luego centrarnos en la base de datos Audio Set. Estudiando los datos hemos podido observar que la base de datos es muy amplia y que abarca una amplia variedad de sonidos distintos, lo que la hace destacar sobre otras bases de datos similares y sin duda permitirá grandes progresos en este problema. Sin embargo Audio Set también tiene varios defectos notables, concretamente la alta heterogeneidad de la repartición de las clases a lo largo de los ejemplos, la limitada compatibilidad con otras librerías de aprendizaje automático que no sean Tensorflow, la dificultad a la hora de introducir los datos en modelos que requieren una cantidad fija de datos de entrada y la falta de especificidad a la hora de describir el modelo base.
Además este TFG se ha centrado en un problema de clasificación poco común como es la clasificación multietiqueta, lo nos ha hecho definir métricas específicas para evaluar la efectividad de los modelos utilizados: la mean Average Precision y el porcentaje de Sonidos Reconocidos.
Finalmente hemos probado la efectividad de varios modelos usando una partición de Audio Set como conjunto de entrenamiento y de evaluación. Hemos elegido como modelos las redes neuronales más comunes: el perceptrón multicapa, la red convolucional y la red LSTM. Las pruebas no han permitido comprobar que las redes recurrentes son más apropiadas para este problema, que el uso de codificación bipolar en las salidas perjudica a la red debido a las adaptaciones que hay que hacer en esta como consecuencia, y que el conjunto de datos utilizado puede afectar a los resultados de forma diferente según el modelo.

6.2 Trabajo futuro

A pesar del trabajo realizado hasta ahora, aún se puede hacer mucho más para profundizar en el estudio de Audio Set y obtener mejores resultados.

En primer lugar, la red LSTM ha proporcionado mejores resultados que otras redes con arquitecturas más complejas. Por ello, si se pretende mejorar los resultados obtenidos la estrategia más adecuada probablemente sea dejar de estudiar perceptrones y redes convolucionales para centrarse en el estudio de las redes recurrentes. Además, al no estar fija la dimensión de los de entrada (concretamente las secuencias de vectores pueden ser de tamaño arbitrario), se pueden usar los datos en formato .tfrecord, los cuales podrían dar lugar a resultados mejores al no recibir los vectores de ceros necesarios en las otras arquitecturas. Otro caso que se podría considerar es ver como afecta el uso del conjunto de entrenamiento no equilibrado a estas redes, caso que no fue considerado en este TFG debido a que el coste potencial de tiempo era demasiado alto. Por otro lado, la red con modelo de atención probabilista debería ser una opción a considerar ya que es la red con mayor mAP actualmente.

En segundo lugar, habría que intentar realizar las pruebas en igualdad de condiciones con las baseline de Google, de forma que se puedan hacer comparaciones consistentes. Esto requeriría por un lado reducir el problema a 485 clases en vez de 527, además de averiguar cuales son las clases que se han retirado y por qué, y por otro buscar más información sobre las métricas definidas por Google y el modelo usado, ya que esta probablemente será las razón principal por la que los resultados de la baseline son tan superiores a todos los obtenidos en nuestras pruebas.

Referencias

[1] “Audio Set, a large-scale dataset of manually annotated audio events”, https://research.google.com/audioset/, 20/05/2018

[2] TEMKO, Andrey, et al. CLEAR evaluation of acoustic event detection and classification systems. En *International Evaluation Workshop on Classification of Events, Activities and Relationships*. Springer, Berlin, Heidelberg, 2006. p. 311-322.

[3] SALAMON, Justin; JACOBY, Christopher; BELLO, Juan Pablo. A dataset and taxonomy for urban sound research. En *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014. p. 1041-1044.

[4] “Detection and Clasification of Acoustic Scenes and Events 2016”, http://cs.tut.fi/sgn/arg/dcase2016/index, , 20/05/2018

[5] GEMMEKE, Jort F., et al. Audio set: An ontology and human-labeled dataset for audio events. En *Acoustics, Speech and Signal Processing (ICASSP), 2017 IEEE International Conference on*. IEEE, 2017. p. 776-780.

[6] KONG, Qiuqiang, et al. Audio Set classification with attention model: A probabilistic perspective. *arXiv preprint arXiv:1711.00927*, 2017.

[7] “Audio Set Users”, https://groups.google.com/forum/#!forum/audioset-users, 20/05/2018

[8] “Keras: The Python Deep Learning Library”, https://keras.io, 20/05/2018

[9] “Tensorflow”, https://www.tensorflow.org

[10] “Cuda”, https://www.nvidia.es/object/cuda-parallel-computing-es.html, 20/05/2018

[11] “Nvidia CuDNN”, https://developer.nvidia.com/cudnn, 20/05/2018

[12] “Perceptrón 5 unidades”, https://es.wikipedia.org/wiki/Perceptr%C3%B3n#/media/File:Perceptr%C3%B3n_5_unidades.svg, 21/05/2018

[13] “Red neuronal convolucional arquitectura”, http://www.diegocalvo.es/wp-content/uploads/2017/07/red-neuronal-convolucional-arquitectura.png, 21/05/2018

[14] “Greff LSTM diagram”, https://deeplearning4j.org/img/greff_lstm_diagram.png, 21/05/2018

Glosario

API	Application Programming Interface
TFG	Trabajo de Fin de Grado
MLP	Multi-Layer Perceptron, Perceptrón Multicapa
CNN	Convolutional Neural Network, Red Neuronal Convolucional
LSTM	Long Short Term Memory
mAP	mean Average Precision
pSR	porcentaje de Sonidos Reconocidos
ReLU	Rectified Linear Unit
GPU	Graphical Processing Unit

Anexos

A Código fuente

Este es el código que ha sido utilizado para las pruebas.

- MLPAudioSet.py:

```
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout
from keras.callbacks import ModelCheckpoint, EarlyStopping
import keras.backend as kb
from os import getcwd
from os.path import join
import numpy as np
import sys, getopt
import h5py

def mavpr( y_true, y_pred ):
        y_pred_r = kb.round( y_pred )
        AP = kb.sum( y_pred_r * y_true, axis = 1 ) / kb.sum( kb.clip( y_pred_r + y_true, 0,
1 ), axis = 1 )
        MAP = kb.mean( AP, axis = 0 )
        return MAP

def mavprbip( y_true, y_pred ):
        return mavpr( ( y_true + 1 ) / 2, ( y_pred + 1 ) / 2 )

def load_data(hdf5_path):
   with h5py.File(hdf5_path, 'r') as hf:
         #x contiene el audio embedding
      x = hf.get('x')
      #y contiene las clases en formato one-hot multiclase
      y = hf.get('y')
      #video_id_list = hf.get('video_id_list')
      #Convertimos los datos a numpy arrays
      x = np.array(x)
      y = np.array(y)
      #video_id_list = list(video_id_list)

   return x, y #, video_id_list

def uint8_to_float32(x):
   return (np.float32(x) - 128.) / 128.

def bool_to_float32(y):
   return np.float32(y)

def main( argv ):
        entradas = 1280
        salidas = 527
```

```
	pathtrain = join( getcwd(), '../Datos/bal_train.h5' )
	pathtest = join( getcwd(), '../Datos/eval.h5' )
	pathModel = '../Modelos/MLP/'
	nameModel = None
	unidadesOcultas = 1500
	capasOcultas = 1
	bip = False
	np.random.seed( 12345 )

	try:
	 opts, args = getopt.getopt( argv, 'hu:c:m:o:b:', [] )
	except getopt.GetoptError:
	 print( 'python MLPAudioSet.py [-h|-u <Unidades ocultas por capa> -c <Capas
ocultas> -m <Lugar donde guardar el modelo> -o <Nombre del modelo> -b <Conversion
bipolar>]' )
	 sys.exit()

	for opt, arg in opts:
	 if opt == '-h':
		print( 'python MLPAudioSet.py [-h|-u <Unidades ocultas por capa> -c
<Capas ocultas> -m <Lugar donde guardar el modelo> -o <Nombre del modelo> -b
<Conversion bipolar>]' )
		sys.exit()
	 elif opt == '-u':
		unidadesOcultas = int( arg )
	 elif opt == '-c':
		capasOcultas = int( arg )
	 elif opt == '-m':
		pathModel = arg
	 elif opt == '-o':
		nameModel = arg
	 elif opt == '-b':
		bip = bool( arg )

	if nameModel == None:
	 nameModel = 'MLP' + str( capasOcultas )
	 if bip:
		nameModel += 'b'
	 nameModel += '.hdf5'
	pathModel = join( getcwd(), pathModel ) + nameModel

	#Creamos el modelo con Keras, un perceptron con una capa oculta
	if bip:
	 act = 'tanh'
	else:
	 act = 'sigmoid'
	model = Sequential()
	model.add( Dense( unidadesOcultas, input_shape=(entradas,), activation='relu' ) )
	model.add( Dropout( 0.3 ) )
	for _ in xrange( capasOcultas - 1 ):
```

```
            model.add( Dense( unidadesOcultas, activation='relu' ) )
            model.add( Dropout( 0.3 ) )
        #Creamos la capa de salida con la activacion correspondiente
        model.add( Dense( salidas, activation=act ) )
        if bip:
            model.compile(loss = 'mse', optimizer='adam', metrics=[mavprbip])
        else:
            model.compile(loss = 'binary_crossentropy', optimizer='adam', metrics=[mavpr])

        print('\n\nCargando datos entrenamiento:')
        (xTrain, yTrain) = load_data(pathtrain)
        #Convertimos los datos de enteros entre 0 y 255 a reales entre -1 y 1, mas faciles de
tratar por las redes
        xTrain = uint8_to_float32(xTrain)      # shape: (N, 10, 128)
        #Aplanamos el embedding para poder usarlo en el perceptron
        xTrain = np.reshape( xTrain, ( -1, 1280 ) )     # shape: (N, 1280)
        print( xTrain.shape )
        #Convertimos los labels a reales
        yTrain = bool_to_float32(yTrain)                # shape: (N, 527)

        print('\n\nCargando datos test:')
        (xTest, yTest) = load_data(pathtest)
        #Convertimos los datos de enteros entre 0 y 255 a reales entre -1 y 1, mas faciles de
tratar por las redes
        xTest = uint8_to_float32(xTest)         # shape: (N, 10, 128)
        #Aplanamos el embedding para poder usarlo en el perceptron
        xTest = np.reshape( xTest, ( -1, 1280 ) )       # shape: (N, 1280)
        print( xTest.shape )
        #Convertimos los labels a reales
        yTest = bool_to_float32(yTest)                  # shape: (N, 527)

        if bip:
            yTrain = yTrain * 2 - 1
            yTest = yTest * 2 - 1

        print('\n\nEntrenando modelo:')
        if bip:
            mc = ModelCheckpoint( filepath=pathModel, monitor='val_mavprbip',
mode='max', verbose=1, save_best_only=True )
            es = EarlyStopping( monitor='val_mavprbip', patience = 3, mode='max' )
        else:
            mc = ModelCheckpoint( filepath=pathModel, monitor='val_mavpr', mode='max',
verbose=1, save_best_only=True )
            es = EarlyStopping( monitor='val_mavpr', patience = 3, mode='max' )
        model.fit( xTrain, yTrain, batch_size = 128, epochs = 50, verbose = 2,
validation_data = ( xTest, yTest ), callbacks = [mc, es] )

if __name__ == '__main__':
        main( sys.argv[1:] )
```

- MLPAudioSetF.py:

```
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout
from keras.callbacks import ModelCheckpoint, EarlyStopping
import keras.backend as kb
from os import getcwd
from os.path import join
import numpy as np
import sys, getopt
import h5py

def mavpr( y_true, y_pred ):
        y_pred_r = kb.round( y_pred )
        AP = kb.sum( y_pred_r * y_true, axis = 1 ) / kb.sum( kb.clip( y_pred_r + y_true, 0,
1 ), axis = 1 )
        MAP = kb.mean( AP, axis = 0 )
        return MAP

def mavprbip( y_true, y_pred ):
        return mavpr( ( y_true + 1 ) / 2.0, ( y_pred + 1 ) / 2.0 )

def load_data(hdf5_path):
   with h5py.File(hdf5_path, 'r') as hf:
      #x contiene el audio embedding
      x = hf.get('x')
      #y contiene las clases en formato one-hot multiclase
      y = hf.get('y')
      #video_id_list = hf.get('video_id_list')
      #Convertimos los datos a numpy arrays
      x = np.array(x)
      y = np.array(y)
      #video_id_list = list(video_id_list)

   return x, y #, video_id_list

def gen_data(hdf5_path, bip):
   with h5py.File(hdf5_path, 'r') as hf:
      #x contiene el audio embedding
      x = hf.get('x')
      #y contiene las clases en formato one-hot multiclase
      y = hf.get('y')
      while True:
         i = 0
         while i < x.shape[0]:
            #Extraemos parte de los datos y los convertimos a numpy arrays
            xT = np.array( x[i:min( i + 128, x.shape[0] )] )
            yT = np.array( y[i:min( i + 128, y.shape[0] )] )
            #Convertimos los datos de enteros entre 0 y 255 a reales entre -1 y 1, mas faciles
de tratar por las redes
            xT = uint8_to_float32(xT) # shape: (N, 10, 128)
```

```
            #Aplanamos el embedding para poder usarlo en el perceptron
            xT = np.reshape( xT, ( -1, 1280 ) )# shape: (N, 1280)
            #Convertimos los labels a reales
            yT = bool_to_float32(yT)            # shape: (N, 527)
            if bip:
                yT= yT * 2 - 1
        i += 128
            perm = np.random.permutation( xT.shape[0] )
            xT = xT[perm]
            yT = yT[perm]
            yield ( xT, yT )

def uint8_to_float32(x):
    return (np.float32(x) - 128.) / 128.

def bool_to_float32(y):
    return np.float32(y)

def main( argv ):
        entradas = 1280
        salidas = 527
        pathtrain = join( getcwd(), '../Datos/unbal_train.h5' )
        pathtest = join( getcwd(), '../Datos/eval.h5' )
        pathModel = '../Modelos/MLP/'
        nameModel = None
        unidadesOcultas = 1500
        capasOcultas = 1
        bip = False
        np.random.seed( 12345 )

        try:
         opts, args = getopt.getopt( argv, 'hu:c:m:o:b:', [] )
        except getopt.GetoptError:
         print( 'python MLPAudioSet.py [-h|-u <Unidades ocultas por capa> -c <Capas
ocultas> -m <Lugar donde guardar el modelo> -o <Nombre del modelo> -b <Conversion
bipolar>]' )
         sys.exit()

        for opt, arg in opts:
         if opt == '-h':
                print( 'python MLPAudioSet.py [-h|-u <Unidades ocultas por capa> -c
<Capas ocultas> -m <Lugar donde guardar el modelo> -o <Nombre del modelo> -b
<Conversion bipolar>]' )
                sys.exit()
         elif opt == '-u':
                unidadesOcultas = int( arg )
         elif opt == '-c':
                capasOcultas = int( arg )
         elif opt == '-m':
                pathModel = arg
```

```
        elif opt == '-o':
                nameModel = arg
        elif opt == '-b':
                bip = bool( arg )

        if nameModel == None:
         nameModel = 'MLP' + str( capasOcultas ) + 'F'
         if bip:
                nameModel += 'b'
         nameModel += '.hdf5'
        pathModel = join( getcwd(), pathModel ) + nameModel

        #Creamos el modelo con Keras, un perceptron con una capa oculta
        if bip:
         act = 'tanh'
        else:
         act = 'sigmoid'
        model = Sequential()
        model.add( Dense( unidadesOcultas, input_shape=(entradas,), activation='relu' ) )
        model.add( Dropout( 0.3 ) )
        for _ in xrange( capasOcultas - 1 ):
         model.add( Dense( unidadesOcultas, activation='relu' ) )
         model.add( Dropout( 0.3 ) )
        #Creamos la capa de salida con la activacion correspondiente
        model.add( Dense( salidas, activation=act ) )
        if bip:
         model.compile(loss = 'mse', optimizer='adam', metrics=[mavprbip])
        else:
         model.compile(loss = 'binary_crossentropy', optimizer='adam', metrics=[mavpr])

        print('\n\nCargando datos test:')
        (xTest, yTest) = load_data(pathtest)
        #Convertimos los datos de enteros entre 0 y 255 a reales entre -1 y 1, mas faciles de
tratar por las redes
        xTest = uint8_to_float32(xTest)         # shape: (N, 10, 128)
        #Aplanamos el embedding para poder usarlo en el perceptron
        xTest = np.reshape( xTest, ( -1, 1280 ) )       # shape: (N, 1280)
        print( xTest.shape )
        #Convertimos los labels a reales
        yTest = bool_to_float32(yTest)                  # shape: (N, 527)

        if bip:
         yTest = yTest * 2 - 1

        print('\n\nEntrenando modelo:')
        if bip:
         mc = ModelCheckpoint( filepath=pathModel, monitor='val_mavprbip',
mode='max', verbose=1, save_best_only=True )
         es = EarlyStopping( monitor='val_mavprbip', patience = 3, mode='max' )
        else:
```

```
        mc = ModelCheckpoint( filepath=pathModel, monitor='val_mavpr', mode='max',
verbose=1, save_best_only=True )
        es = EarlyStopping( monitor='val_mavpr', patience = 3, mode='max' )
        #Calculamos el numero de batches por epoca
        with h5py.File(pathtrain, 'r') as hf:
         x = hf.get('x')
         spe = np.ceil( x.shape[0] / 128.0 )

        model.fit_generator( gen_data( pathtrain, bip ), steps_per_epoch = spe, epochs = 50,
validation_data = ( xTest, yTest ), callbacks = [mc, es] )

if __name__ == '__main__':
        main( sys.argv[1:] )
```

- CNNAudioSet.py:

```
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout, Conv2D, MaxPooling2D, Flatten
from keras.callbacks import ModelCheckpoint, EarlyStopping
import keras.backend as kb
from os import getcwd
from os.path import join
import numpy as np
import sys, getopt
import h5py

def mavpr( y_true, y_pred ):
        y_pred_r = kb.round( y_pred )
        y_true_r = kb.round( y_true )
        AP = kb.sum( y_pred_r * y_true_r, axis = 1 ) / kb.sum( kb.clip( y_pred_r + y_true_r,
0, 1 ), axis = 1 )
        MAP = kb.mean( AP, axis = 0 )
        return MAP

def load_data(hdf5_path):
   with h5py.File(hdf5_path, 'r') as hf:
      #x contiene el audio embedding
      x = hf.get('x')
      #y contiene las clases en formato one-hot multiclase
      y = hf.get('y')
      #video_id_list = hf.get('video_id_list')
      #Convertimos los datos a numpy arrays
      x = np.array(x)
      y = np.array(y)
      #video_id_list = list(video_id_list)

   return x, y #, video_id_list

def uint8_to_float32(x):
   return (np.float32(x) - 128.) / 128.
```

```
def bool_to_float32(y):
   return np.float32(y)

def main( argv ):
        entradas = (10, 128, 1)
        salidas = 527
        pathtrain = join( getcwd(), '../Datos/bal_train.h5' )
        pathtest = join( getcwd(), '../Datos/eval.h5' )
        pathModel = join( getcwd(), '../Modelos/CNN' )
        nameModel = 'CNN.hdf5'
        np.random.seed( 12345 )

        try:
         opts, args = getopt.getopt( argv, 'hm:o:', [] )
        except getopt.GetoptError:
         print( 'python MLPAudioSet.py [-h|-m <Lugar donde guardar el modelo> -o
<Nombre del modelo>]' )
         sys.exit()

        for opt, arg in opts:
         if opt == '-h':
                print( 'python MLPAudioSet.py [-h|-m <Lugar donde guardar el modelo> -o
<Nombre del modelo>]' )
                sys.exit()
         elif opt == '-o':
                nameModel = arg
         elif opt == '-m':
                pathModel = join( getcwd(), arg )

        pathModel = join( pathModel, nameModel )

        #Creamos el modelo con Keras, una red convolucional
        model = Sequential()
        model.add( Conv2D( 16, ( 3, 1 ), input_shape=entradas, activation='relu' ) )
        model.add( MaxPooling2D() )
        model.add( Dropout( 0.3 ) )
        model.add( Flatten() )
        model.add( Dense( 600, activation='relu' ) )
        model.add( Dense( salidas, activation='sigmoid' ) )
        model.compile(loss='binary_crossentropy', optimizer='adam', metrics=[mavpr])

        print('\n\nCargando datos entrenamiento:')
        (xTrain, yTrain) = load_data(pathtrain)
        #Convertimos los datos de enteros entre 0 y 255 a reales entre -1 y 1, mas faciles de
tratar por las redes
        xTrain = uint8_to_float32(xTrain)      # shape: (N, 10, 128)
        xTrain = np.expand_dims( xTrain, -1 )
        print( xTrain.shape )
        #Convertimos los labels a reales
        yTrain = bool_to_float32(yTrain)                # shape: (N, 527)
```

```
        print('\n\nCargando datos test:')
        (xTest, yTest) = load_data(pathtest)
        #Convertimos los datos de enteros entre 0 y 255 a reales entre -1 y 1, mas faciles de tratar por las redes
        xTest = uint8_to_float32(xTest)         # shape: (N, 10, 128)
        xTest = np.expand_dims( xTest, -1 )
        print( xTest.shape )
        #Convertimos los labels a reales
        yTest = bool_to_float32(yTest)                  # shape: (N, 527)

        print('\n\nEntrenando modelo:')
        mc = ModelCheckpoint( filepath=pathModel, verbose=1, save_best_only=True, monitor='val_mavpr', mode='max' )
        es = EarlyStopping( monitor='val_mavpr', patience = 3, mode='max' )
        #es = EarlyStopping( patience = 3 )
        model.fit( xTrain, yTrain, batch_size = 128, epochs = 50, verbose = 2, validation_data = ( xTest, yTest ), callbacks = [mc, es] )

if __name__ == '__main__':
        main( sys.argv[1:] )
```

- LSTMAudioSet.py:

```
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout, LSTM
from keras.callbacks import ModelCheckpoint, EarlyStopping
import keras.backend as kb
from os import getcwd
from os.path import join
import numpy as np
import sys, getopt
import h5py

def sonrec( y_true, y_pred ):
        y_pred_r = kb.round( y_pred )
        y_true_r = kb.round( y_true )
        sr = kb.sum( y_pred_r * y_true_r )
        st = kb.sum( y_true_r )
        return sr / st

def mavpr( y_true, y_pred ):
        y_pred_r = kb.round( y_pred )
        y_true_r = kb.round( y_true )
        AP = kb.sum( y_pred_r * y_true_r, axis = 1 ) / kb.sum( kb.clip( y_pred_r + y_true_r, 0, 1 ), axis = 1 )
        MAP = kb.mean( AP, axis = 0 )
        return MAP

def load_data(hdf5_path):
   with h5py.File(hdf5_path, 'r') as hf:
```

```
        #x contiene el audio embedding
        x = hf.get('x')
        #y contiene las clases en formato one-hot multiclase
        y = hf.get('y')
        #video_id_list = hf.get('video_id_list')
        #Convertimos los datos a numpy arrays
        x = np.array(x)
        y = np.array(y)
        #video_id_list = list(video_id_list)

    return x, y #, video_id_list

def uint8_to_float32(x):
    return (np.float32(x) - 128.) / 128.

def bool_to_float32(y):
    return np.float32(y)

def main( argv ):
        entradas = (10, 128)
        salidas = 527
        pathtrain = join( getcwd(), '../Datos/bal_train.h5' )
        pathtest = join( getcwd(), '../Datos/eval.h5' )
        pathModel = join( getcwd(), '../Modelos/LSTM' )
        nameModel = 'LSTM.hdf5'
        np.random.seed( 12345 )

        try:
         opts, args = getopt.getopt( argv, 'hm:o:', [] )
        except getopt.GetoptError:
         print( 'python MLPAudioSet.py [-h|-m <Lugar donde guardar el modelo> -o
<Nombre del modelo>]' )
         sys.exit()

        for opt, arg in opts:
         if opt == '-h':
                print( 'python MLPAudioSet.py [-h|-m <Lugar donde guardar el modelo> -o
<Nombre del modelo>]' )
                sys.exit()
         elif opt == '-o':
                nameModel = arg
         elif opt == '-m':
                pathModel = join( getcwd(), arg )

        pathModel = join( pathModel, nameModel )

        #Creamos el modelo con Keras, una red LSTM
        model = Sequential()
        model.add( LSTM( 600, input_shape = entradas ) )
        model.add( Dropout( 0.3 ) )
```

```
        model.add( Dense( 600, activation='relu' ) )
        model.add( Dense( salidas, activation='sigmoid' ) )
        model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=[mavpr,sonrec])

        print('\n\nCargando datos entrenamiento:')
        (xTrain, yTrain) = load_data(pathtrain)
        #Convertimos los datos de enteros entre 0 y 255 a reales entre -1 y 1, mas faciles de
tratar por las redes
        xTrain = uint8_to_float32(xTrain)      # shape: (N, 10, 128)
        print( xTrain.shape )
        #Convertimos los labels a reales
        yTrain = bool_to_float32(yTrain)               # shape: (N, 527)

        print('\n\nCargando datos test:')
        (xTest, yTest) = load_data(pathtest)
        #Convertimos los datos de enteros entre 0 y 255 a reales entre -1 y 1, mas faciles de
tratar por las redes
        xTest = uint8_to_float32(xTest)        # shape: (N, 10, 128)
        print( xTest.shape )
        #Convertimos los labels a reales
        yTest = bool_to_float32(yTest)                 # shape: (N, 527)

        print('\n\nEntrenando modelo:')
        mc = ModelCheckpoint( filepath=pathModel, verbose=1, save_best_only=True,
monitor='val_mavpr', mode='max' )
        #es = EarlyStopping( monitor='val_mavpr', patience = 3, mode='max' )
        model.fit( xTrain, yTrain, batch_size = 128, epochs = 50, verbose = 2,
validation_data = ( xTest, yTest ), callbacks = [mc] )


if __name__ == '__main__':
        main( sys.argv[1:] )
```

- cargarModelo.py:

```
import keras
import keras.backend as kb
from os import getcwd
from os.path import join
import numpy as np
import sys, getopt
import h5py

def mavpr( y_true, y_pred ):
        y_pred_r = kb.round( y_pred )
        AP = kb.sum( y_pred_r * y_true, axis = 1 ) / kb.sum( kb.clip( y_pred_r + y_true, 0,
1 ), axis = 1 )
        MAP = kb.mean( AP, axis = 0 )
        return MAP
```

```
def mavprbip( y_true, y_pred ):
	return mavpr( ( y_true + 1 ) / 2, ( y_pred + 1 ) / 2 )

def mavprN( y_true, y_pred ):
	AP = np.sum( y_pred * y_true, axis = 1 ) / np.sum( np.clip( y_pred + y_true, 0, 1 ),
axis = 1 )
	MAP = np.mean( AP )
	return MAP


def sonrec( y_true, y_pred ):
	sr = np.sum( y_pred * y_true )
	st = np.sum( y_true )
	return float( sr ) / st

def load_data(hdf5_path):
   with h5py.File(hdf5_path, 'r') as hf:
      #x contiene el audio embedding
      x = hf.get('x')
      #y contiene las clases en formato one-hot multiclase
      y = hf.get('y')
      #video_id_list = hf.get('video_id_list')
      #Convertimos los datos a numpy arrays
      x = np.array(x)
      y = np.array(y)
      #video_id_list = list(video_id_list)

   return x, y #, video_id_list

def uint8_to_float32(x):
   return (np.float32(x) - 128.) / 128.

def main( argv ):
	path = '../Modelos/MLP/MLP1.hdf5'
	ev = '../Datos/eval.h5'
	bip = False
	modo = 1

	try:
	 opts, args = getopt.getopt( argv, 'hm:e:b:d:', [] )
	except getopt.GetoptError:
	 print( 'python cargarModelo.py [-h|-m <Modelo a cargar> -e <Datos a evaluar> -b
<Conversion a bipolar> -d <modo>]' )
	 sys.exit()

	for opt, arg in opts:
	 if opt == '-h':
		print( 'python cargarModelo.py [-h|-m <Modelo a cargar> -e <Datos a
evaluar> -b <Conversion a bipolar> -d <modo>]' )
		sys.exit()
```

```
        elif opt == '-m':
                path = arg
        elif opt == '-e':
                ev = arg
        elif opt == '-b':
                bip = bool( arg )
        elif opt == '-d':
                modo = int( arg )

    if bip:
        co = { 'mavprbip' : mavprbip }
    else:
        co = { 'mavpr' : mavpr }
    model = keras.models.load_model( path, custom_objects = co )

    (x, yTrue) = load_data( ev )
    if modo == 1:
        x = uint8_to_float32( x ).reshape( -1, 1280 )
    elif modo == 3:
        x = np.expand_dims( uint8_to_float32( x ), -1 )
    yTrue = np.float32( yTrue )

    yPred = model.predict( x )
    if bip:
        yPred = ( yPred + 1 ) / 2
    yPred = yPred.round()

    print( yPred.sum() )
    print( 'Media de sonidos reconocidos: ' + str( sonrec( yTrue, yPred ) ) )
    print( 'AP media: ' + str( mavprN( yTrue, yPred ) ) )

if __name__ == '__main__':
    main( sys.argv[1:] )
```